2013-09-13

Behavior Models and Composition for Software and Systems Architecture

Auguston, Mikhail

Auguston, Mikhail, Clifford Whitcomb, "Behavior Models and Composition for Software and Systems Architecture", ICSSEA 2012, 24th International Conference on SOFTWARE &

Behavior Models and Composition for Software and Systems Architecture

Mikhail Auguston

Department of Computer Science, Naval Postgraduate School
Monterey, CA 93943, USA
Phone: 1-831-656-2607, fax: 1-831-656-2814, email: maugusto@nps.edu

Clifford Whitcomb

Department of Systems Engineering, Naval Postgraduate School
Monterey, CA 93943, USA
Phone: 1-831-656-3001, fax: 1-831-656-2814, email: cawhitco@nps.edu

Abstract: This paper suggests an approach to formal software and systems architecture specification based on behavior models. The behavior of a system is defined as a set of events (event trace) with two basic relations: precedence and inclusion. The structure of an event trace is specified using event grammars and other constraints organized into schemas. Behaviors for both system and its environment can be specified within the same framework. Suggested composition operations on schemas are based on event pattern matching and provide for behavior merging and abstract interface specification. The schema framework is amenable to stepwise refinement, reuse, visualization of multiple architecture views, and application of automated tools for consistency checks and system behavior verification early in the design process.

Key words: Software and system architecture, executable architecture models

1. INTRODUCTION

The concept of system architecture has emerged in the last two decades as one of the fundamental concepts in software and systems engineering. ISO 2011 [20] defines architecture as the "fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution", and plays a key role as a structural basis of a system-of-interest for respective stakeholders. The architecture plays a role as the bridge between requirements and implementation of a system. The following aspects have emerged as characteristic for architecture descriptions [30], [11].

- An architecture description belongs to a high level of abstraction, ignoring many of the implementation details, such as algorithms and data structures.
- An architecture specification should be supportive for the refinement process, and needs to be checked carefully at each refinement step (preferably with tools).
- There should be flexible and expressive composition operations for the refinement process.
- The architecture specification should support the reuse of well-known architectural styles and patterns. Practice has provided several well-established, reusable architectural solutions.
- An architecture of a system should be considered in the context of the environment in which it operates, as suggested in the international standard ISO/IEC IEEE 42010 "Systems and Software Engineering Architecture Description" [20].
- The software architect needs a number of different views of the software architecture for the various uses and users [23](including visual representations, like diagrams).

Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to satisfy the requirements and serve as a basis for the design [30]. An architecture description has converged on the concept of architectural elements, such as component, connector, and relationships among them. "When designers discuss or

present a software architecture for a specific system, they typically treat the system as a collection of interacting components. Components define the primary computations of the application. The interactions or connections between components define the ways in which the components communicate or otherwise interact with each other." [1]. A conclusion in [33] states: "Every system has an architecture, whether or not it is documented and understood."

Correct behavior of the system is the main concern for the system's developers. Using a computer to solve a problem usually requires finding an algorithm and mapping it on the appropriate computational platform, i.e. designing a behavior by applying a step-by-step procedure to solve the problem at hand. System architects may be interested in verifying a system's interaction with the operational environment, e.g. by querying a system model to find scenarios that contain potential hazard states. System of Systems architects are concerned with the emergent behaviors resulting from the interactions of subsystems. Errors in the early system design are the most expensive to fix when detected later in the development lifecycle.

All these considerations suggest the importance of architecture models, and the practical need to test and verify the system architecture early in the design phase. Behavior modeling is at the core of our approach. We suggest software and systems architecture modeling framework called *Monterey Phoenix* (or MP) based on the following principles:

- A view on the architecture as a high level description of possible system behaviors, emphasizing the behavior of subsystems and interactions between subsystems.
- Concurrency of actions is a default, unless ordering is imposed (thus representing a design decision introducing a dependency between activities).
- The importance of specifying the interaction between the system and its environment. A model of the system and its environment behaviors and interactions can be a contribution to the system's requirements specification.
- The event grammar provides a view of the behavior as a set of actions (event trace) with two basic relations, where the PRECEDES relation captures the dependency abstraction, and the IN relation represents the hierarchical relationship. Since the event trace is a set of events, additional constraints can be specified using set-theoretical operations and predicate logic.
- The behavior composition operations, which support architecture reuse and refinement towards design and implementation models.
- The MP architecture description is amenable to deriving different views, including a structural view (traditional architecture box-and-arrow diagrams) or those desired by the Department of Defense Architecture Framework (DoDAF) [15].
- Executable architecture models provide the possibility to automatically generate examples of behaviors (use cases) for early system architecture testing and verification with tools.

MP models may be used for early design testing and verification, early performance and safety assessment estimates, and for generating examples of scenarios (use cases), which in turn can be used to support test case construction and monitoring for system implementation testing. MP architecture models can be integrated into standard frameworks, like UML, SysML, DoDAF, providing the level of abstraction convenient for architecture models with the emphasis on behavior and interaction aspects (see Example 7 in sec. 4.4 for more details).

2. BRIEF RELATED WORK SURVEY

The following ideas of behavior modeling and formalization have provided inspiration and insights for this work.

Literate programming introduced by D.Knuth set the directions for hierarchical refinement of structure mapped into behavior, with the concept of pseudo-code and tools to support the refinement process [22].

Campbell and Habermann [14] and Bruegge and Hibbard [13] have demonstrated the application of path expressions for program monitoring and debugging. Path expressions in [30] have been used (semi-formally) as a part of software architecture description.

A.Hoare's CSP (Communicating Sequential Processes) [18], [32] is a framework for process modeling and formal reasoning about those models. This behavior modeling approach has been applied to software architecture descriptions to specify a connector's protocol [2], [3], [29].

Rapide [25], [26] uses events and partially ordered sets of events (posets) to characterize component interaction.

D.Harel's Statecharts [17] became one of the most common behavior modeling frameworks, integrated in broader modeling and specification systems UML [12], and AADL [16].

Wang and Parnas [36] have proposed to use trace assertions to formally specify the externally observable behavior of a software module and presented a trace simulator to symbolically interpret the trace assertions and simulate the externally observable behavior. The approach is based on algebraic specifications and term rewriting.

The Alloy modeling framework [21] has strongly influenced this work through ideas of integration of sets and first order predicate logic within the relational logic framework, inheritance structure, emphasis on lightweight Formal Methods as opposed to the full-scale theorem proving, with the fundamental concept of Small Scope Hypothesis, and the principles of immediate feedback and visualization during model design.

The concept of software behavior models based on events and event traces was introduced in [4], [5], [6], [7] as an approach to software debugging and testing automation. The early draft of Monterey Phoenix has appeared in [8], [9].

3. BEHAVIOR MODELS

In a certain sense, software is a compact description for a set of required behaviors. The source code in any programming language – a finite object by itself – specifies a potentially infinite number of execution paths. The behavior of the system is usually the main concern for the developer, and the presence of unintended behaviors manifests errors in the design. A system is operating in a certain environment, which has its own behavior that interacts with the system. The main objective of our approach is to provide a formal framework for specifying behaviors of the system, its parts and environment, and the interaction between them.

3.1 Event concept

An implemented software system usually represents an algorithm, i.e. a step-by-step description of activities tailored towards achieving a certain goal. The MP behavior model is based on the concept of an *event* as an abstraction of activity. The event has a beginning and an end, and may have duration (a time interval during which the action is accomplished). The behavior of a system is modeled as a set of events with two binary relations defined for them: precedence (PRECEDES) and inclusion (IN) – the *event trace*. One action is required to precede another if there is a dependency between them, e.g. the Send event should precede the Receive event. Events may be nested, when a complex activity contains a set of other activities. Imposing one of these basic relations on a pair of activities represents an important design decision. Usually system behavior does not require a total ordering of events. Both PRECEDES and IN are partial ordering relations. If two events are not ordered, they may occur concurrently. Appendix 1 provides more details specifying the properties of the basic relations.

3.2 Event grammar

The structure of possible event traces is specified by an event grammar. A grammar rule specifies structure for a particular event type (in terms of IN and PRECEDES relations). A grammar rule has form

A: right-hand-part;

where A is an event type name. Event types that do not appear in the left hand part of rules are considered atomic and may be refined later by adding corresponding rules. More details about event grammar notation can be found in [8]. We'll provide just a short description illustrated by examples here.

Sequence denotes the ordering of events under the PRECEDES relation. Events can be composed to describe possible event traces by using in the right hand part of the event grammar rule the following composition operations:
ordered sequence of events **A B C**, alternative **(A | B | C)**, ordered iteration **(* A B C *)** (A B C repeated zero or more times), **(+ A B C +)** (one or more times), **[A] -** optional element, **{A, B, C} -** set of unordered (potentially concurrent) events, **{* A *}** - set of zero or more of unordered events A, **{+ A +}** - set of one or more of unordered events. An event grammar is essentially a graph grammar, which specifies directed acyclic graphs of events with the arcs representing relations IN and PRECEDES.

Example 1. An event grammar for car race scenarios.

```
car_race:       {+ driving_a_car +};
driving_a_car:  go_straight  (*  ( go_straight | turn_left | turn_right )  *)  stop;
go_straight:    ( accelerate | decelerate | cruise );
```

Similar to context-free grammars, event grammars can be used as production grammars to generate instances of event traces. An instance of event trace satisfying the grammar can be visualized as a directed graph with two types of edges (one for each of the basic relations).

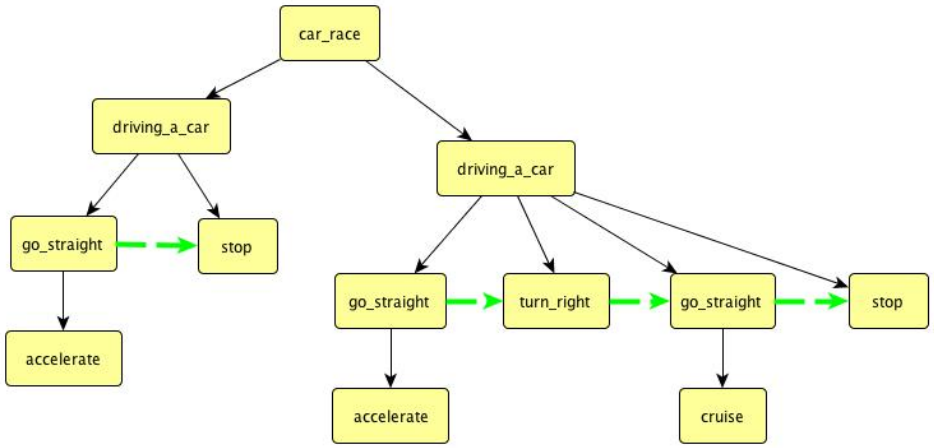

Fig. 1. An event trace derived from the event grammar in Example 1.

4. BEHAVIOR COMPOSITION AND ARCHITECTURE VIEWS

The behavior of a particular system is specified as a set of all possible event traces using a *schema*. The concept of Monterey Phoenix schema has been inspired by the Z schema [34]. The purpose is to define the structure of all possible event traces (in terms of IN and PRECEDES relations) using event grammar rules and other constraints. A schema usually contains a collection of events called *roots* representing the behaviors of parts of the system (e.g. components and connectors in common architecture descriptions), *composition operations* specifying interactions between these behaviors, and additional constraints on root behaviors.

There is precisely one instance of each root event in any trace. The schema also may contain auxiliary grammar rules defining composite event types used in other rules. Roots in turn may be defined as schemas, thus providing for architecture reuse and composition. A schema may define both finite and infinite traces, but most analysis tools for reasoning about a system's behavior assume that a trace is finite.

The schema represents instances of behavior (event traces), in the same sense as Java source code represents instances of program execution. Just as a particular program execution path can be extracted from a Java program's source code by running it on a JVM, a particular event trace specified by a MP schema can be generated from the event grammar rules by applying behavior composition operations and constraints.

Example 2. (a simple pipe/filter architecture pattern).

```
SCHEMA simple_message_flow
ROOT Task_A:    (* send *);
ROOT Task_B:    (* receive *);
COORDINATE      (* $x: send  *)     FROM Task_A,
                (* $y: receive *)   FROM Task_B     ADD $x PRECEDES $y;
```

In order to establish coordination between sending and receiving messages, we use the behavior composition operation **COORDINATE**. In this example the composition operation takes two traces and defines a modified event trace (merges behaviors of Task_A and Task_B) by adding the **PRECEDES** relation between the selected **send** and **receive**.

The first part of composition operation (the source) uses event patterns to specify segments of root traces that should be selected. The **(* $x: send *)** pattern identifies the sequence of totally ordered **send** events (with respect to the transitive closure of **PRECEDES** relation – **PRECEDES***). Use of the **(* P *)** pattern for selection means that all events P in the source root should be ordered, both iterations should have the same number of selected elements (**send** events from the first trace and **receive** events from the second), and the pair selection follows this ordering (*synchronous coordination*). Labels **$x** and **$y** provide access to the events selected within each iteration. The **ADD** composition completes the behavior adjustment, specifying that an ordering relation will be imposed on each pair of selected events. Behavior specified by this schema is a set of matching event traces for Task_A and Task_B with the modifications imposed by the composition.

The composition operation may be considered as an abstract interface description for root behaviors. In the case when *asynchronous coordination* is needed, an iterative set pattern can be used. For example,

```
COORDINATE {* $x: E1  *} FROM A, {* $y: E2 *} FROM B ADD $x PRECEDES $y;
```

In this case matching root traces for A and B still should contain an equal number of selected events of types E1 and E2, correspondingly. But now the resulting merged traces will include all permutations of events E2 from B matching events E1 from A, with the **PRECEDES** relation imposed on each selected pair. This assumes that other constraints, like the partial ordering axioms from Appendix 1, are satisfied. Each permutation yields one potential instance of a resulting trace for the schema deploying this composition. In order to reduce the exponential explosion, optimizations similar to symmetry reduction in model checking tools should be considered. Changing **(* … *)** for **{*… *}** in Example 2 may increase the number of composed traces in the schema.

Different views for different stakeholders can be extracted from MP schemas. For example, each root may be visualized as a box, and if there is a composition operation specifying an interaction between root behaviors, the boxes are connected by an arrow marked by the interaction type. The root behavior may be visualized with UML Activity Diagrams [12] (see Example 7). The MP developer's environment may have a library of predefined views providing different visualizations for schemas.

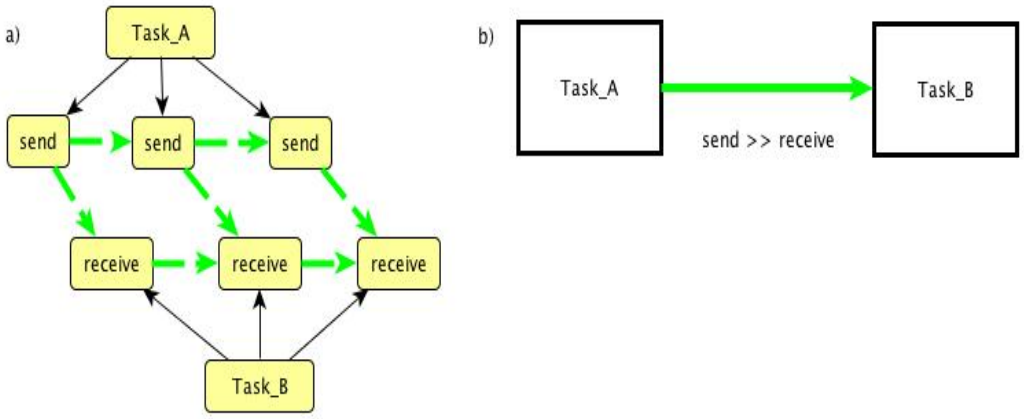

Fig. 2. a) Example of composed event trace for the simple_message_flow schema.
b) An architecture view for the simple_message_flow schema.

4.1 Data items as behaviors

Data items in MP are represented by actions (events) that may be performed on that data. This principle follows the ADT concept introduced in [24].

Example 3. Data flow.

```
SCHEMA Data_flow
ROOT Process_1: (* work write *);
ROOT Process_2: (* ( read | work ) *);
ROOT File: (* write *) (* read *);
Process_1, File SHARE ALL write;
Process_2, File SHARE ALL read;
```

Behavior of the File requires that all **write** operations should be completed before any **read** operations. The **SHARE ALL** composition ensures that the schema admits only event traces where corresponding event sharing is implemented. Event sharing is in fact yet another way of behavior coordination (similar to the *rendezvous* in Ada). It is assumed that shared events may appear in the root event at any level of nesting. The view of this schema in Fig.3 b) renders root interaction with a line where the shared event name is attached as a label.

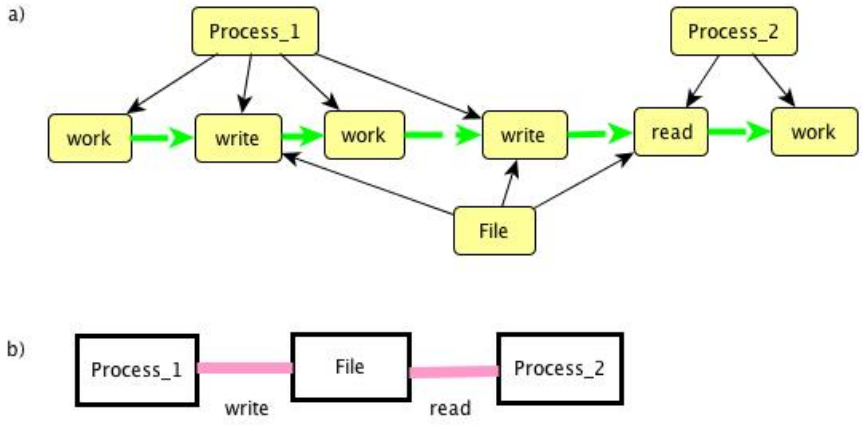

Fig. 3. a) An example of composed event trace for the Data_flow schema.
b) An architecture view for the Data_flow schema.

4.2 Reuse of schemas

Example 4. Stack behavior.

```
SCHEMA Stack
ROOT  Stack_operation: (*  ( push | pop )  *);
SATISFIES FOREACH $x: pop FROM Stack_operation
                ( Number_of (pop) before ($x) < Number_of (push) before ($x) );
```

This schema specifies the behavior of a stack in terms of stack primitive operations. Let **IN*** denote the transitive closure of the **IN** relation (similarly as **PRECEDES*** is a transitive closure for **PRECEDES**). The domain of the universal quantifier is the set of all **pop** events **e**, such that **(e IN* Stack_operation).** The function **Number_of (pop) before ($x)** yields the number of **pop** events **e** such that **(e PRECEDES* $x)**. The set of event traces specified by this schema contains only traces that satisfy the constraint. This example presents a filtering operation as yet another kind of behavior composition.

Example 5. Reuse of a schema.

```
SCHEMA Two_stacks_in_use
INCLUDE Stack;
ROOT Main: {* (do_something | use_S1 | use_S2) *};
  use_S1:  (push | pop) ;
  use_S2:  (push | pop) ;
ROOT S1: Stack;
ROOT S2: Stack;
S1, Main SHARE ALL $x: (pop | push) SUCH THAT Has_enclosing (use_S1)($x) WITHIN Main;
S2, Main SHARE ALL $x: (pop | push) SUCH THAT Has_enclosing (use_S2)($x) WITHIN Main;
```

The **INCLUDE** statement brings the schema Stack into the scope. This means that all constraints specified in the Stack also will be included. The rule for Main is intentionally left lax without imposing any specific ordering on embedded activities. Roots **S1** and **S2** represent the presence of two independent stacks as data items. The ordering of **pop** and **push** events inside **use_S1** and **use_S2** in each stack behavior is ensured and will be brought into the resulting trace by the included Stack behaviors as a result of sharing these events with the Stack behavior. The **SHARE ALL** composition operation uses event patterns and context conditions to accomplish the necessary event trace construction. The predicate **Has_enclosing(T)(e1)** is true iff there exists an event e2 of the type T in the trace specified by the **WITHIN** clause, such that **e1 IN* e2**.

Predicates and functions like **Has_enclosing(T)(e**), and **Number_of (T) before (e)** are used for convenient navigation in the event graphs.

4.3 Components and connectors

Connectors and components, which are core elements in an architecture description, can be uniformly modeled in MP as behaviors. The idea that connectors should be elevated to the first-class-citizen status on a par with components is often discussed in the literature, for example, in [35].

Suppose that the communication between the components is implemented via a buffer of size **max_buffer_size**, and not necessarily all sent messages are consumed, i.e. some of them could stay in the buffer indefinitely. Each message may be consumed no more than once, and the ordering of receiving does not necessarily correspond to the ordering of sending. The root **Buffered_channel** simulates the behavior of a connector between **Task_A** and **Task_B**. This behavior model does not provide details about what happens after a buffer overflow event.

Example 6.

```
SCHEMA Buffered_transaction
ROOT Task_A:: (* Send *);
ROOT Task_B:: (* Receive *);
ROOT Buffered_channel: {*  (Send  [ Receive ] )  *}   (Overflow | Normal);

Task_A, Buffered_channel SHARE ALL Send;
Task_B, Buffered_channel SHARE ALL Receive;
SATISFIES FOREACH $x: Receive FROM Buffered_channel
      ( Number_of (Send) before ($x) - Number_of (Receive) before ($x) ) <= max_buffer_size;
SATISFIES FOREACH $x: Overflow FROM Buffered_channel
      ( Number_of (Send) before ($x) - Number_of (Receive) before ($x) ) > max_buffer_size;
SATISFIES FOREACH $x: Normal FROM Buffered_channel
      ( Number_of (Send) before ($x) - Number_of (Receive) before ($x) ) <= max_buffer_size;
```

If the schema should satisfy only behaviors without buffer overflow, the three **SATISFIES** conditions above can be replaced by the following constraint (and the **Overflow** event can be removed from the schema):

```
SATISFIES FOREACH $x: Send FROM Buffered_channel
    Number_of ($y: Send) before ($x) SUCH THAT ( ¬ Has_next(Receive)($y) )   <  max_buffer_size;
```

Note that **PRECEDES** relation is defined explicitly either in the grammar rule, or by **ADD** composition operation, and is a proper subset of its transitive closure **PRECEDES***. The predicate **Has_next(T)(e1)** is true iff there exists an event **e2** of the type **T** in the trace, such that **e1 PRECEDES e2**.

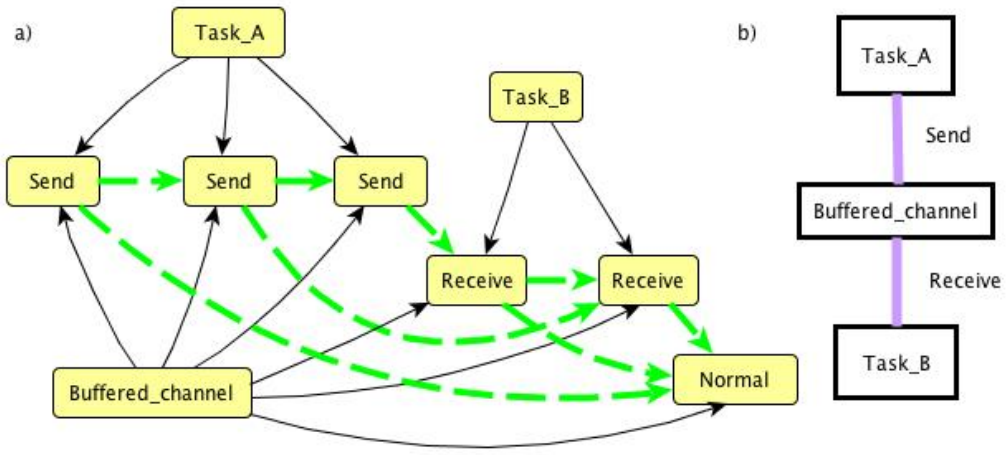

Fig. 4. a) An example of event trace (without overflow) for the Buffered_transaction schema with max_buffer_size = 3.
b) An architecture view for the Buffered_transaction schema.

4.4 Environment's behavior

The following example demonstrates how to integrate the behavior of an environment with the behavior of a system. The ATM_withdrawal schema specifies a set of possible interactions between the Customer, ATM_system, and Data_Base. An event trace generated from this schema can be considered as a use case example.

Example 7. Withdraw money from ATM.

```
SCHEMA ATM_withdrawal
ROOT Customer:      (*  insert_card
                        ( (  identification_succeeds
                             request_withdrawal
                           ( get_money | not_sufficient_funds ) )  |
                          identification_fails                      )    *);
ROOT ATM_system:  (*    read_card          validate_id
                        ( id_successful    check_balance
                                ( (sufficient_balance      dispense_money) |
                                  unsufficient_balance     )                    |
                          id_failed                                             )    *);
ROOT Data_Base:     (* ( validate_id | check_balance ) *);

Data_Base,  ATM_system SHARE ALL validate_id, check_balance ;

COORDINATE     (* $x: insert_card  *)        FROM Customer,
               (* $y: read_card *)          FROM ATM_system        ADD $x PRECEDES $y ;
COORDINATE     (* $x: request_withdrawal *) FROM Customer,
               (* $y: check_balance *)      FROM ATM_system        ADD $x PRECEDES $y ;
COORDINATE     (* $x: identification_succeeds *) FROM Customer,
               (* $y: id_successful *)  FROM ATM_system            ADD $y PRECEDES $x ;
COORDINATE     (* $x: get_money  *) FROM Customer,
               (* $y: dispense_money *) FROM ATM_system            ADD $y PRECEDES $x ;
COORDINATE     (* $x: not_sufficient_funds *) FROM Customer,
               (* $y: unsufficient_balance *) FROM ATM_system      ADD $y PRECEDES $x ;
COORDINATE     (* $x: identification_fails  *) FROM Customer,
               (* $y: id_failed  *)          FROM ATM_system      ADD $y PRECEDES $x ;
```

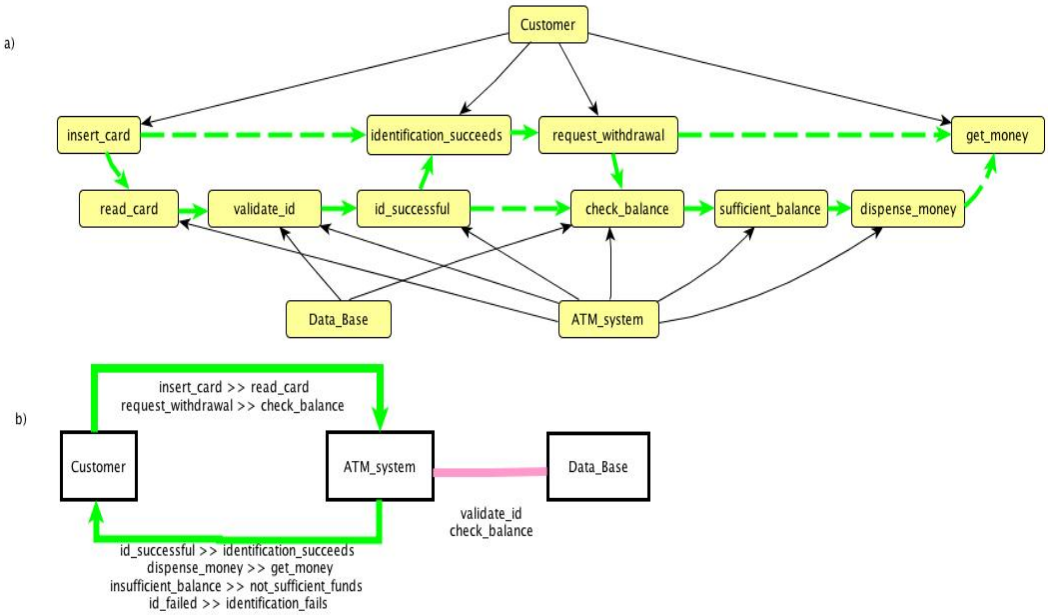

Fig. 5. a) An example of event trace for the ATM_withdrawal schema.
b) An architecture view for the ATM_withdrawal schema.

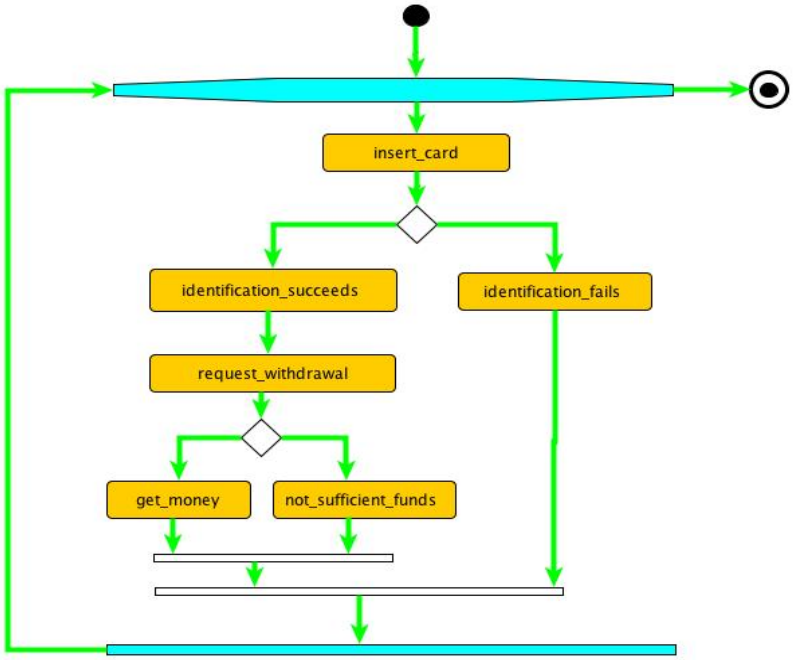

Fig. 6. A view on the Customer root event behavior as an Activity Diagram.

If the view of the whole system's behavior emphasizing the interaction between the parts (components) can be visualized as in Fig. 5, b), the view of root's standalone behavior can be visualized as an UML Activity Diagram (Fig.6 provides an example for the Customer root behavior). Since event aggregates (iterations, alternatives, sets) in MP are well structured, it is possible to use Nassi–Shneiderman diagrams [28] as yet another kind of view. The event trace on Fig. 5, a) can be viewed as an analog of UML Sequence Diagram's "swim lanes" for the Customer and ATM_system interaction. This example demonstrates that MP models can be integrated into standard frameworks, like UML, SysML, DoDAF, providing the level of abstraction convenient for architecture models, where, in particular, MP focuses on the interaction aspects.

4.5 Merging schemas

So far, we have seen examples of assembling schemas using previously defined schemas (Example 5). Each schema in the assembly holds its own roots and composition operations (**SATISFIES** filter, and interaction constraints, like **COORDINATE** and **SHARE ALL**) within its scope.

The join operation for schemas looks like:

SCHEMA A EXTENDS B
Roots for A
Constraints and composition operations involving roots from both A and B

The resulting schema A joins roots defined in A and roots defined in B, merges within its scope all constraints and composition operations defined in B, and may have additional constraints and composition operations involving all roots. Appendix 1 contains **Base** schema specifying properties for basic relations **IN*** and **PRECEDES***. It is assumed that any MP schema extends on **Base**. This operation on schemas is inspired by the Z schema expressions concept [34].

A typical use of such schema composition may be for assembling the architecture of a System-of-systems from the architectures of its constituent systems.

5. ASSERTIONS AND QUERIES

An event trace represents an example of particular execution of the system (or use case, especially if the behavior of the environment is included) that can evolve from the architecture specified by a schema. Event traces can be effectively generated from the event grammar rules and then adjusted and filtered according to the composition operations in the schema. This justifies the term *executable architecture model* for MP. It is possible to obtain all valid event traces within a certain limit. Usually such a limit (*scope*) may be set by the maximum total number of events within the trace, or by the upper limit on the number of iterations in grammar rules (recursion in the grammar rules can be limited in similar ways). For many purposes a modest limit of a maximum 3 iterations will be sufficient. This process of generating and inspecting event traces for the schema is similar to the traditional software testing process.

In the case of MP models it is possible to automatically generate all event traces within the given scope (exhaustive testing). Careful inspection of generated traces (scenarios/use cases) may help developers identify undesired behaviors. Usually it is easier to evaluate an example of behavior (particular event trace) than the generic description of all behaviors (the schema). The *Small Scope Hypothesis* [21] states that most errors can be demonstrated on relatively small counterexamples.

Certain properties of behavior can be formalized as assertions about traces (similar to the **SATISFIES** constraint in Example 4 and Example 6 above), and verified exhaustively for all event traces within the scope, yielding the counterexamples when the assertion is violated. For example, hazard states can be specified as a result of certain interactions between the system and its environment, and then the traces within scope can be searched for a trace that matches the hazard scenario. An example of such assertion checking performed on a MP prototype is given in [10]. Since assertion checking is performed on a complete event trace, it becomes possible to refer to events following a given event, for example, to specify fairness conditions. This brings the expressiveness of MP assertions closer to temporal logic [31].

In a similar fashion queries can be performed on the traces, providing different kinds of statistics. For example, events may have *attributes*, such as estimated duration, and system's performance estimates can be obtained from collecting a representative amount of event traces and calculating durations for event sets of interest.

Another example of an event attribute may be the probability of an events in alternatives, like **(A [0.3] | B [0.7])** establishing that **A** happens with the probability 0.3 and **B** with probability 0.7. Now it becomes possible to estimate probabilities of certain event traces, e.g. probability for the system to get into a hazard state. This opens a whole direction for system simulation and statistical experiments based on executable systems architecture models and their environment models.

6. IMPLEMENTATION PROTOTYPES

The first MP prototype [10] has been implemented as a compiler generating an Alloy model [21] from the MP schema and then running the Alloy Analyzer to obtain event traces and to perform assertion checks. It has benefited from Alloy's relational logic formalism and visualization tools. Performance depends on the performance of SAT solver used by Alloy Analyzer.

Direct trace generation from the event grammar can be accomplished quite efficiently, and the process of generating all traces for the given schema and within a given scope can be roughly described by the following procedure.

1. For each root in the schema generate all possible traces within the given scope.
2. Select one trace from each root's collection. Apply all the schema's composition operations and filters. If the resulting composed trace is consistent with the schema's filters and composition operations, it is included into the schema trace collection. Otherwise, proceed with the next selection.

This process may lead to an exponential explosion, but it has potential for optimization by applying early pruning whenever possible. The main optimization ideas stem from the considerations that composition

operations (**COORDINATE** and **SHARE ALL**) usually require an equal number of selected events in the matching traces. Root traces can be sorted according to the number of required events to avoid selection of inconsistent root traces in Step 2. Careful rearrangement of composition operations and filters may also provide a significant speed up in the trace assembly.

We have built a prototype trace generator by converting MP schemas into a C++ code, and then compiling and running it. This architecture solution is similar to one that has been implemented, for instance, in the SPIN/PROMELA model checker (using C as a target language) [19].

Several optimizations similar to the mentioned above have been implemented. A sample run on an iMac with 2.8 GHz/4 GB yields the following performance for a schema example with approximately 60 lines of MP source text, 31 event types, including 9 roots, 10 composite event types, 12 atomic event types, 12 **SHARE ALL** compositions, and for a maximum scope of 3 for iterations (actually it is an architecture model for the MP -> C++ prototype itself).

Total 1328 traces generated, with total 79836 events, average 60.1175 events/trace, max trace length 69;
Initial search space (number of all root trace selections before filtering) 35100;
Selection ratio 3.78348%, generation speed 18021.8 events/sec;
Elapsed time (including compilation of the generated C++ code) 4.42997 sec.

7. CONCLUSIONS

The MP executable architecture models provide a high level of abstraction for testing, verifying, and documenting system architecture early in the design phase. The main advantages may be summarized as follows.

- The use of MP focuses the attention of developers early on the behavior of the system and provides tools to verify the assumptions.
- The schema framework is amenable to stepwise architecture refinement, reuse, composition, visualization, and application of automated tools for consistency checks.
- The executable architecture models integrated with the environment behavior models can be helpful for identifying emerging behaviors.
- The ability to generate use cases for requirements specification and for testing the system's implementation.
- The ability to create abstract views on the interfaces, composition, and coordination within the system.
- The ability to develop performance estimates based on statistics obtained from the event traces.
- The possibility to extract different architecture views, for example based on stakeholder viewpoints, from the architecture model.

7.1 What is next?

Architecture modeling touches on the very fundamental issues in systems engineering and software design processes, and has substantial consequences for the next phases in software system design. There are many threads of future research based on the ideas described above.

- Monitoring whether the behavior of implemented system matches the MP architecture model (testing automation). If the source code of implementation can be marked up to indicate which segments of code start and end corresponding MP events, it becomes possible to log actual execution traces, and to check them for consistence with expected behaviors.
- Developing methods and techniques for early performance, throughput, and latency estimates based on duration and frequency estimates for events within components and connectors.

- Developing methods and techniques for an architecture model's static analysis, for example, by verifying MP models with a model checking tool.
- Introducing architecture metrics for MP models for system cost estimates.
- Development of a library of reusable architecture patterns.
- Development of a library of reusable architecture views.
- Development of a collection of reusable environment behavior models, including business process models in MP.
- Extending the MP approach to the meta-architecture level to support software product lines, and domain-specific architectures by representing the variation points as macro-conditions in schemas. The same mechanism may be used for architecture configuration management.

APPENDIX 1

Base specifies a filter for every event trace and ensures that it satisfies partial order axioms for **IN*** and **PRECEDES*** relations. It uses predefined generic event type **Event**. The special variable **$Trace** stands for the whole trace specified by a schema. The purpose of this schema is similar to the purpose of virtual class in OO paradigm.

SCHEMA Base
-- there are no roots, this schema is used only to bring the following filter into derived schema
SATISFIES FOREACH $a, $b, $c: Event FROM $Trace

-- Mutual Exclusion of Relations

($a PRECEDES* $b ⇒ ¬($a IN* $b)) ∧ *-- Axiom 1)*
($a PRECEDES* $b ⇒ ¬($b IN* $a)) ∧ *-- Axiom 2)*
($a IN* $b ⇒ ¬($a PRECEDES* $b)) ∧ *-- Axiom 3)*
($a IN* $b ⇒ ¬($b PRECEDES* $a)) ∧ *-- Axiom 4)*

-- Non-commutativity

($a PRECEDES* $b ⇒ ¬($b PRECEDES* $a)) ∧ *-- Axiom 5)*
($a IN* $b ⇒ ¬($b IN* $a)) **∧** *-- Axiom 6)*

-- Irreflexivity for PRECEDES and IN* follows from non-commutativity.*

-- Transitivity

(($a PRECEDES* $b) ∧ ($b PRECEDES* $c) ⇒ ($a PRECEDES* $c)) **∧** *-- Axiom 7)*
(($a IN* $b) ∧ ($b IN* $c) ⇒ ($a IN* $c)) **∧** *-- Axiom 8)*

-- Distributivity

(($a IN* $b) ∧ ($b PRECEDES* $c) ⇒ ($a PRECEDES* $c)) ∧ *-- Axiom 9)*
(($a PRECEDES* $b) ∧ ($c IN* $b) ⇒ ($a PRECEDES* $c)); *-- Axiom 10)*

Each MP schema uses **Base** as a default extension. As a result, each schema will filter its event traces accordingly, for example, the following schema has an empty set of traces, because it violates Axiom 5 for partial ordering.

SCHEMA Wrong EXTENDS Base
ROOT A: a b;
ROOT B: b a;
A, B SHARE ALL a, b;

ACKNOWLEDGMENTS

The authors would like to thank Monica Farah-Stapleton, Joey Rivera, and Timothy Shields for valuable feedback. The diagrams in Fig. 1 - 6 have been designed with the yEd Graph Editor:
http://www.yworks.com/en/products_yed_about.html

REFERENCES

[1] ABOWD, G., ALLEN, R., GARLAN, D., 1995, Formalizing Style to Understand Descriptions of Software Architecture, ACM Transactions on Software Engineering and Methodology 4(4): 319-364

[2] ALLEN, R., 1997, A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, CMU Technical Report CMU-CS-97-144, May 1997

[3] ALLEN, R., GARLAN, D., 1997, A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology, Vol. 6(3): 213-249, July 1997.

[4] AUGUSTON, M., 1991, FORMAN - Program Formal Annotation Language, in Proceedings of 5th Israel Conference on Computer Systems and Software Engineering, Herclia, May 27-28, IEEE Computer Society Press, 1991, pp.149-154.

[5] AUGUSTON, M., 1995, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, in Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, May 1995.

[6] AUGUSTON, M., JEFFERY, C., UNDERWOOD, S., 2002, A Framework for Automatic Debugging, in Proceedings of the 17th IEEE International Conference on Automated Software Engineering, September 23-27, 2002, Edinburgh, UK, IEEE Computer Society Press, pp.217-222.

[7] AUGUSTON, M., MICHAEL, B., SHING, M., 2006, Environment Behavior Models for Automation of Testing and Assessment of System Safety, Information and Software Technology, Elsevier, Vol. 48, Issue 10 , October 2006, pp. 971-980

[8] AUGUSTON, M., 2009, Software Architecture Built from Behavior Models, ACM SIGSOFT Software Engineering Notes, 34:5.

[9] AUGUSTON, M., 2009, Monterey Phoenix, or How to Make Software Architecture Executable, OOPSLA'09/Onward conference, OOPSLA Companion, October 2009, pp.1031-1038

[10] AUGUSTON, M., WHITCOMB, C., 2010, System Architecture Specification Based on Behavior Models, in Proceedings of the 15th ICCRTS Conference (International Command and Control Research and Technology Symposium), Santa Monica, CA, June 22-24, 2010

[11] BASS, L.; CLEMENTS, P., KAZMAN, R., 2003, *Software Architecture In Practice*, 2nd Edition, Boston, Addison-Wesley.

[12] BOOCH, G., JACOBSON, I., RUMBAUGH, J., 2000, OMG Unified Modeling Language Specification, http://www.omg.org/docs/formal/00-03-01.pdf

[13] BRUEGGE, B., HIBBARD, P., 1983, Generalized Path Expressions: A High-Level Debugging Mechanism, The Journal of Systems and Software 3, 1983, pp. 265-276.

[14] CAMPBELL, R.H., HABERMANN, A.N., 1974, The Specification of Process Synchronization by Path Expressions, Lecture Notes in Computer Science, No. 16, Apr. 1974, pp. 89-102.

[15] Department of Defense Architecture Framework, 2010, http://dodcio.defense.gov/dodaf20.aspx

[16] FEILER, P., GLUCH, D., HUDAK, J., The Architecture Analysis & Design Language (AADL): An Introduction, Technical Note CMU/SEI-2006-TN-011,

[17] HAREL, D., 1987, A Visual Formalism for Complex Systems. Science of Computer Programming 8(3), pp.231–274

[18] HOARE, C. A. R., Communicating Sequential Processes. Prentice-Hall, 1985.

[19] HOLZMANN, G., 2004, The SPIN Model Checker, Boston, Addison-Wesley

[20] ISO 2011, International Organization for Standardization. ISO Standard ISO/IEC 42010:2007, "Systems and Software Engineering – Recommended Practice for Architectural Description of Software-Intensive Systems."

[21] JACKSON, D., 2006, Software Abstractions: Logic, Language, and Analysis. Cambridge, Massachusetts: The MIT Press.

[22] KNUTH, D., 1984, Literate Programming, The Computer Journal, 27(2): 97-111, May 1984

[23] KRUCHTEN, P., 1995, Architectural Blueprints - the 4+1 View Model of Software Architecture. IEEE Software. 12 (6), pp. 42-45

[24] LISKOV, B., ZILLES, S., 1974, Programming with abstract data types, ACM SIGPLAN Notices, Vol 9 Issue 4, pp. 50 – 59

[25] LUCKHAM, D., AUGUSTIN, L., KENNEY, J., VERA, J., BRYAN, D., MANN, W. 1995, Specification and analysis of system architecture using Rapide. IEEE Transactions on Software Engineering, Special Issue on Software Architecture, 21(4): 336–355, April 1995.

[26] LUCKHAM, D., J., VERA, J., 1995, An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, 21(9): 717–734, September 1995.

[27] MEDVIDOVIC, N., ROSENBLUM, D., REDMILES, D., 2002, Modeling Software Architectures in the Unified Modeling Language, ACM Transactions on Software Engineering and Methodology, Vol.11, No. 1, January 2002, pp.2-57.

[28] NASSI, I.; SHNEIDERMAN, B., 1973, Flowchart techniques for structured programming, ACM SIGPLAN Notices XII, August 1973, pp.12 – 26

[29] PELLICCIONE, P., INVERARDI, P., MUCCINI, H., 2009, CHARMY: A Framework for Designing and Verifying Architectural Specifications, IEEE Transactions on Software Engineering, Vol. 35, No 3, 2009, pp.325-346

[30] PERRY, D., WOLF, A., 1992, Foundations for the Study of Software Architecture, ACM SIGSOFT Software Engineering Notes, 17:4, pp. 40-52.

[31] PNUELI, A., 1981, A temporal logic of programs, Theoretical Computer Science, 13: pp.45-60.

[32] ROSCOE, B., 1997, The Theory and Practice of Concurrency, Prentice Hall International Series in Computer Science (580pp), ISBN 0-13-674409-5

[33] ROZANSKI, N., WOODS, E., 2012. Software Systems Architecture, 2nd Edition, Addison-Wesley.

[34] SPIVEY, J.M., The Z Notation: A reference manual, Prentice Hall International Series in Computer Science, 1989. (2nd Ed., 1992)

[35] TAYLOR, R., MEDVIDOVIC, N., DASHOFY, E., 2010, Software Architecture, Foundations, Theory, and Practice, John Wiley & Sons, Inc.

[36] WANG, Y., PARNAS, D., 1994, Simulating the behavior of software modules by trace rewriting, IEEE Trans. Software Eng. 20, 10 (Oct. 1994), pp. 750-759.